'Revised 5/84

# Internal Representation of Text and Graphics

Objectives of Module

In this module you will learn how to manipulate text and graphics on your screen without using BASIC statements to print or plot. You will discover how to place information directly into memory so as to change what appears on the screen. This understanding will enable you to create dramatic graphics and custom designed character sets.

Overview (subtopics)

1. Color Registers.
   An overview of the relationship between the different graphics modes and the five color registers.

2. What is Screen RAM?
   Explores the purpose of screen RAM and its use with different graphics modes.

3. Hi-Res Graphics Mode 8.
   What are pixels, bytes, and bits and how are they related in Graphics mode 8?

4. Graphics Modes 3 through 7.
   The special coding of colors in these graphics modes is explained.

5. Graphics Modes 0, 1, and 2.
   Why characters look the way they do and how the computer decides what character to put on the screen and what color to make it.

6. Summary and Challenges.

Prerequisite Understanding Necessary

1. You must be familiar with how to use the different BASIC graphics modes to put text or graphics on your screen.

2. You must know the purpose of the PEEK and POKE statements and how to use them in BASIC.

Materials Needed

1. BASIC Cartridge.

## Color Registers

This section explores the use of the color registers in different graphics modes and reviews how BASIC uses these registers.

Computer memory can be thought of as a sequence of mailboxes with each mailbox having its own address. The address of the first mailbox in computer memory is zero, the second has an address of one, and the last has an address of 65535 (assuming the computer has 64K of memory). Each mailbox holds a byte. A byte is a number no smaller than zero and no bigger than 255. This number, or byte, can be used for many different purposes. Many bytes in memory are reserved for special purposes.

There are five special bytes whose purpose is to control the colors you see on the screen. Their addresses are 708, 709, 710, 711, and 712. Each of these locations is called a color register (numbered 0 through 4). Locations 708 through 711 are also often referred to as playfields 0 through 3 repectively.

Internal Representation Worksheet #1 will help you understand the relatonship between the value put into a color register, the color you see, and the numbers used in the SETCOLOR command supplied by BASIC. Turn to that worksheet now.

## <u>Internal Representation Worksheet #1</u>

Most of the screen in Graphics mode 0 (the background) is called playfield 2. The color of this playfield comes from color register 2 which is memory location 710. If you POKE a new value between 0 and 255 into location 710, the color of the screen will change. Try the following:

```
POKE 710,0
POKE 710,15
POKE 710,16
```

Now press SYSTEM RESET. Pressing SYSTEM RESET will reload memory location 710 with the default value of 148 for the background color of the Graphics 0 screen.

You can also change the color of playfield 2 by using the SETCOLOR command. The form of this command is:

```
     SETCOLOR register,hue,luminance
```

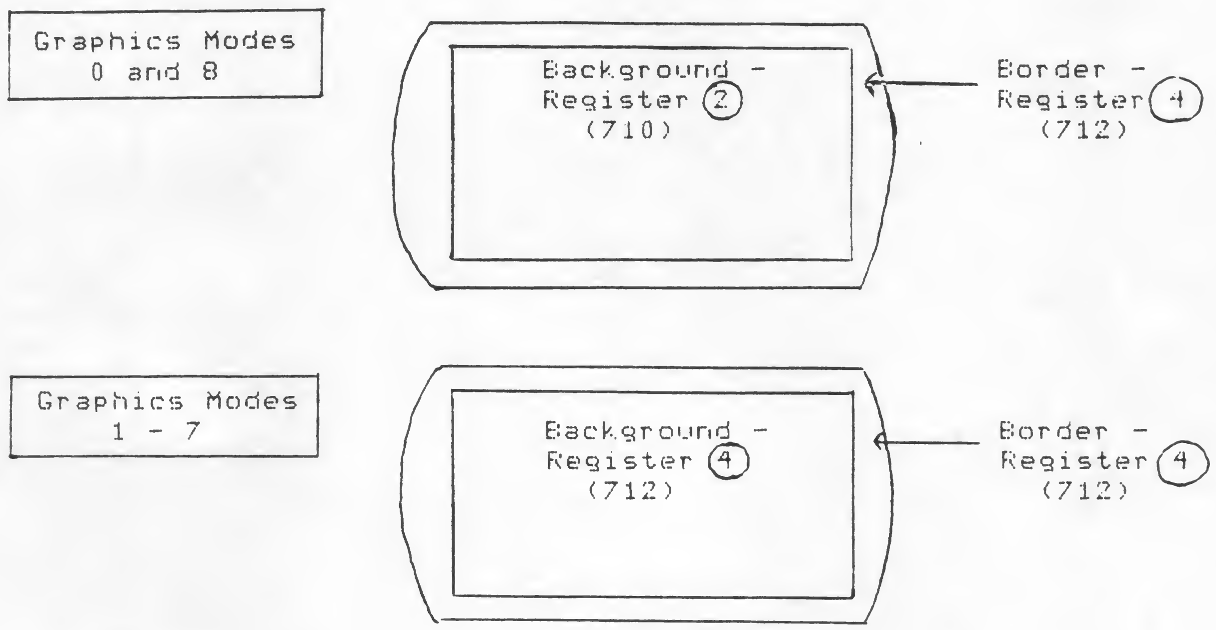
The color "registers" are numbered 0 to 4. The "hue" and "luminance" are numbers from 0 to 15. All this command really does is change the value in the appropriate color register. Type in the SETCOLOR and PRINT commands listed below to see what value is stored in the color register and fill in the chart.

| | Hue | Lum | Value in Color Register |
|---|---|---|---|
| SETCOLOR 2,0,0<br>PRINT PEEK(710) | 0 | 0 | |
| SETCOLOR 2,0,15<br>PRINT PEEK(710) | 0 | 15 | |
| SETCOLOR 2,1,0<br>PRINT PEEK(710) | 1 | 0 | |
| SETCOLOR 2,1,15<br>PRINT PEEK(710) | 1 | 15 | |
| SETCOLOR 2,5,0<br>PRINT PEEK(710) | 5 | 0 | |
| SETCOLOR 2,15,15<br>PRINT PEEK(710) | 15 | 15 | |

Can you guess what the relationship is between the hue and luminance of a color, and the value in the corresponding color register? Consider dividing the value in the color register by 16 to get the hue. <u>Before you proceed, be sure you figure this out or ask someone for help</u>.

In worksheet #1 you used color register 2 (location 710) to change the background color of the screen. One of the confusing aspects of Atari color graphics is that the color of the background is controlled by register 2 only in certain graphics modes (0 and 8). In most graphics modes the background color comes from register 4 (location 712). This can get very confusing, especially when you consider that in all graphics modes the border color comes from register 4 also. Diagram 1 might help clarify this.

Diagram 1


Graphics Modes 0 and 8
Background - Register 2 (710)
Border - Register 4 (712)
Graphics Modes 1 - 7
Background - Register 4 (712)
Border - Register 4 (712)

Now the interesting thing is that since the background and border get their color from the same register in modes 1 through 7, one can never make the background a different color from the border in these modes. The entire screen will always be the color in register 4. Try POKEing different values into locations 710 and 712 in different graphics modes to better understand the diagrams above. Also, try using "SETCOLOR 2,hue,lum" and "SETCOLOR 4,hue,lum".

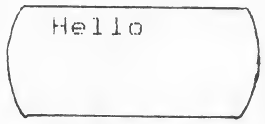
## What is Sreen RAM?

In this section you will learn about an area of memory called screen RAM which is used to store information that appears on your screen as text or graphics.

Like all other memory locations, screen RAM is just a series of bytes used for a special purpose. Each byte is a value from 0 to 255. The reason these memory locations are called screen RAM is because everything you see on your screen is the result of the computer looking at each number in screen memory and putting whatever corresponds to that number on the screen. What is displayed depends on which graphics mode you are using. The numbers in screen RAM and interpreted differently for each graphics mode. It makes absolutely no difference whether you use BASIC, or PILOT, or Assembly Language to display text or graphics on your screen. In every case, when something is displayed on the screen, there are numbers in screen RAM which correspond to what you see. Similarly, if there's something in screen RAM besides zeros, then there will be something on your monitor. The interesting part of all of this is that the same numbers are used in screen memory, 0 to 255, yet these numbers can provide distinctly different displays on your screen in different graphics modes. The diagram below demonstrates how numbers in screen RAM correspond to what is displayed on the screen.

Screen RAM: ... 40 101 108 108 111 ...

In Graphics 0 this is decoded into:



In other graphics modes, these very same numbers in screen RAM would produce a very different display on the screen. Later in this module you will learn how the computer decides what to put on the screen. For now, however, you should complete Worksheet #2 to get familiar with the idea of screen RAM.

## Internal Representation Worksheet #2

On the Atari, screen RAM is not restricted to a particular area in the computer's memory, rather it can be almost anywhere you want it to be. Usually one doesn't have to worry about it though, because BASIC takes care of it whenever you PRINT or PLOT to the screen. But the computer always needs a way to find out where screen RAM is so that it can look there to decide what to show on the screen. So, when BASIC sets aside an area of memory for screen RAM, it also saves an address so it can locate screen RAM when you PRINT or draw on the screen. That address is stored in memory locations 88 and 89.

By playing a little with PEEKs and POKEs, you can do some interesting things to screen RAM to make different things happen on the screen. Start by following these steps:

1. Press the SHIFT and CLEAR keys simultaneously to clear the screen. The cursor should be positioned at the top of the screen. It will be automatically positioned two spaces in from the left-most edge.

2. On this top line type your name. Don't press RETURN. Use the CTRL and the arrow keys to move the cursor down a couple of lines and to the left margin (2 spaces in from the screen edge).

3. Be sure you are in the capital letters mode by pressing the SHIFT and CAPS keys.

Now type: `LOC=256*PEEK(89)+PEEK(88)` and press RETURN.

You have just assigned the address of the current screen RAM location to a variable called LOC.

4. The first two locations in screen RAM are blank because your name is indented by two spaces. Thus you will see two zeros if you type:

```
PRINT PEEK(LOC),PEEK(LOC+1)
```

5. Try typing the following lines. Always press RETURN after each line.

```
PRINT PEEK(LOC+2),PEEK(LOC+3)
POKE LOC+20,PEEK(LOC+2)
```

Notice what happens on the top line of the screen. You just POKEd the number stored in LOC+2 of screen RAM, which happened to represent the first letter of your name, and you stored a copy of it in LOC+20, another screen RAM location. Now try this next line.

```
POKE LOC+40,PEEK(LOC+3)
```

6. Play with the ideas above for a while to explore the possibilities. For example, try things like:

POKE LOC+40,165. Try other numbers.

You can also explore screen RAM in other graphics modes, but first you need to find out where screen RAM is located after you enter the graphics mode. Type the following exactly as it is shown below, without line numbers.

```
GRAPHICS 3
LOC=256*PEEK(89)+PEEK(88)
```

Now try some of the following:

```
POKE LOC,255
POKE LOC,39
FOR I=0 TO 255:POKE LOC,I:FOR J=1 TO 80:NEXT J:NEXT I
FOR I=0 TO 255:POKE LOC+I,I:FOR J=1 TO 20:NEXT J:NEXT I
```

Repeat these same steps starting with declaring the graphics mode and assigning the screen RAM address to LOC, but do it in Graphics 7 this time. The numbers POKEd into memory are the same, yet the display is quite different. Experiment with poking different values into screen RAM on your own now.

WARNING: Locations 88 and 89 (address of screen RAM) are only used by BASIC so that it knows where to print characters or draw graphics. Unfortunately, the Antic chip in the computer looks somewhere else to find the location of the memory that contains information that should be displayed on the screen. These locations occur in the display list and are discussed in the Display Lists module.

## Hi-Res Graphics Mode 8

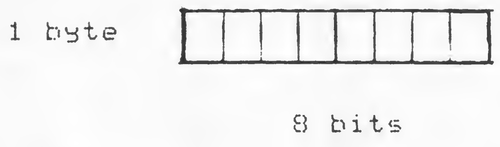
In this section you will learn how the numbers in screen RAM enable the computer to know where to turn on dots of light on your Graphics mode 8 screen.

Before you can understand how the computer internally represents the graphics on your screen, you must have a complete understanding of bytes, bits, and the binary (base 2) number system.

If your computer has 48K of RAM, that means there are 48K, or 48 x 1024, cells of random access memory. A byte is simply a number which each of these memory cells can hold. These numbers can only be in the range of 0 to 255.

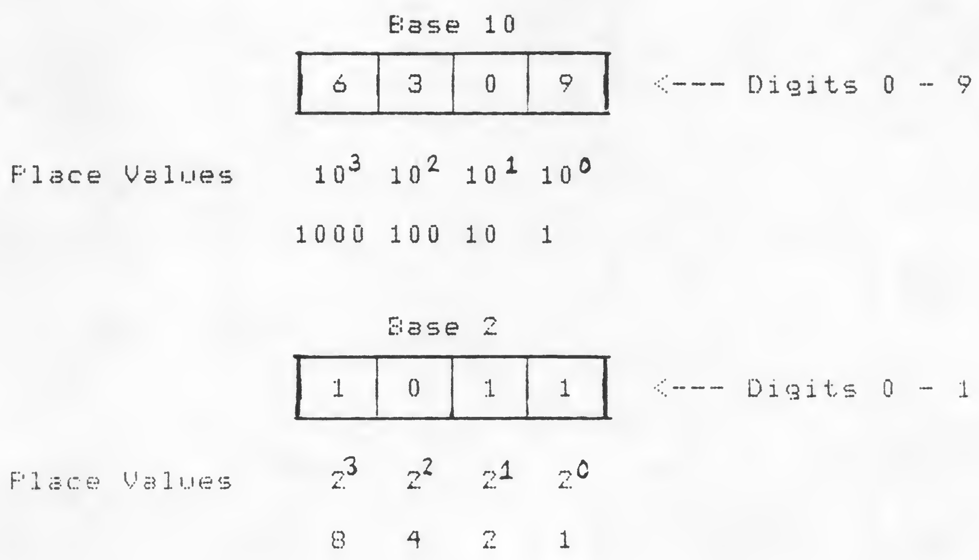
The numbers placed in computer memory called bytes are made up of 8 smaller units called bits (binary digits). Each bit is either a zero or a one -- zero and one are the two digits used in the binary number system.

Diagram 2



In base ten there are 10 digits used: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The number 746 in base ten has a 6 in the ones place (6 x 1) and a 4 in the tens place (4 x 10) and a 7 in the hundreds place (7 x 100). In base ten each position has a place value. Base two follows the same principles. Study Diagram 3 below. Note that the exponents for each place value in the two numbering systems are the same.

## Diagram 3



The largest number a byte can be is 255. This is not just some arbitrary number. When all 8 bits in a byte are ones, the sum of the place values is 255. Try to complete Worksheet #3.

## Internal Representation Worksheet #3

1. Try filling in the blanks in the following problem which converts a binary number into its decimal equivalent.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Binary Number | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| Place Value | 128 | ___ | ___ | ___ | 8 | 4 | 2 | 1 |

```
Decimal Equivalent = 1 x 128 +
                     0 x ___ +
                     0 x ___ +
                    __ x ___ +
                    __ x  8  +
                    __ x ___ +
                    __ x ___ +
                    __ x ___
                               = 150
```

If you POKEd the number 150 into a memory location, you would really be storing the binary number 10010110 (one byte) in that memory location.

2. With a BASIC cartridge in the computer, type the following statements:

```
GRAPHICS 8
SETCOLOR 2,0,0
SCRAM=256*PEEK(89)+PEEK(88)
SCRAM=SCRAM+80:POKE SCRAM,0
```

Remember, the computer looks at the beginning of screen RAM, the address of which is stored in locations 88 and 89, to find the information that goes in the upper left-hand corner of the screen. The variable SCRAM was assigned the first location of screen RAM. Then a zero was POKEd into screen RAM at location SCRAM + 80. There is no noticeable change on the screen because a zero represents a blank space when it is stored in screen RAM.

Now experiment with poking some other values into screen RAM. Use the keyboard CTRL and arrow keys to move the cursor up to the line: SCRAM=SCRAM+80:POKE SCRaM,0. Place the cursor over the zero at the end of the line and change it to each of the decimal numbers in the list below. Each time you change the value, press RETURN.

| Decimal Number to Poke | Binary Equivalent |
|---|---|
| 255 | 11111111 |
| 240 | 11110000 |
| 15 | 00001111 |
| 204 | 11001100 |
| 51 | 00110011 |
| 128 | 10000000 |
| 64 | 01000000 |
| 32 | 00100000 |
| 16 | 00010000 |
| 8 | 00001000 |
| 4 | 00000100 |
| 2 | 00000010 |
| 1 | 00000001 |

Everytime you execute the statement SCRAM=SCRAM+80:POKE SCRAM,0, SCRAM gets changed. This is so that when you POKE a new number into screen RAM it gets put in a different place on the screen so that you do not erase the numbers you previously put into memory. In fact, by adding 80 each time to SCRAM, the numbers you POKE into screen memory are displayed two lines below on the screen. The reason for this will be explained later.

Try to figure out the relationship between the little bits of light that appear on the screen and the numbers you POKEd into screen memory. Look at the binary representation of each number and notice how the bits that are 1's correspond to the spots that light up. Ignore any colors you see -- that's simply a feature of your TV or monitor called artifacting.

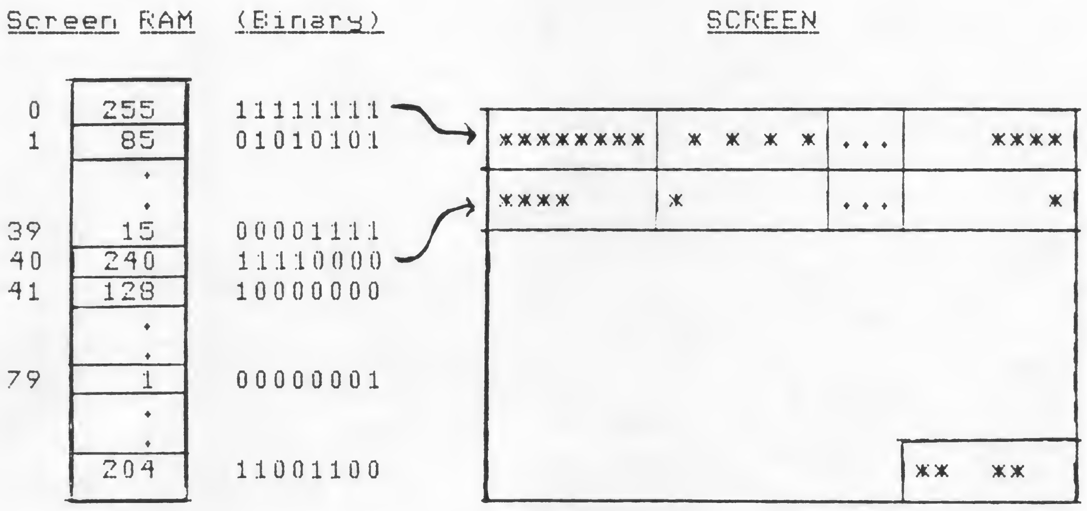
Try other numbers and see if you can predict what will appear on the screen.

NOTE: Notice that the biggest binary number that we can store in a memory location is 11111111. This equals 255 in decimal and that is why we can't store a bigger number in one memory location. If you would like more practice with the binary number system, see the Number Systems and Conversion section of the Machine Architecture module.

Each single dot of light that appeared on your screen in Worksheet #3 is called a pixel. Pixels are very tiny in Graphics mode 8 which is why this mode is called a high resolution mode. Pixels are much larger in other graphics modes.

Hopefully you realized that in Graphics mode 8 each bit in a byte determines whether a pixel is on or off. For every bit that is a one, the corresponding pixel on the screen is turned on (see Diagram 4).

Diagram 4



How many pixels of light will Graphics 8 allow across the screen? The answer is 320. This is called the horizontal resolution of a graphics screen. Let's do some calculations:

Each of the 8 bits in a byte corresponds to one pixel and so we can calculate ...

8 bits per byte x _?_ bytes = 320 bits (pixels).
So the number of bytes is 40.

Screen RAM is simply a sequential list of bytes. In Graphics 8 the first 40 bytes in this list are used to turn on the appropriate pixels in the first row across the top of the screen. The next 40 bytes (SCRAM+40 to SCRAM+79) are used for the second row, and so on. This is why everytime 80 was added to SCRAM in Worksheet #3, it simply moved our graphics down two rows when a number was POKEd into that location.

Finally, how much screen RAM is needed for an entire Graphics 8 screen? The vertical resolution (number of rows) in Graphics 8 is 192. Thus you need 40 bytes per row x 192 rows = 7680 bytes. That's a large amount of memory (almost 8K).

## Graphics Modes 3 Through 7

The four-color graphics modes 3, 5, and 7 are explored here and the methods in which numbers in screen RAM are used to put colors on the screen are explained. You will also learn why graphics modes 4 and 6 are provided when in fact Graphics 5 and 7 do everything Graphics 4 and 6 do, and Graphics 5 and 7 provide more colors.

In Graphics 8 each bit in a byte is used to tell the computer whether to turn on or turn off a pixel on the screen. Unfortunately this coding scheme provides no way to use color registers - all you get is an on or off pixel.

In modes 3, 5, and 7 each pixel can be any one of three colors or the color of the background. That's why these are called four-color graphics modes. In order to accomplish this, a different scheme than that used with Graphics 8 had to be devised. Rather than each bit in a byte corresponding to a pixel on the screen, every two bits in a byte corresponds to a pixel.

Any two bits can be used to provide the computer with up to four possible values:

| | | | |
|---|---|---|---|
| 0 = $\boxed{0}\boxed{0}$ | 1 = $\boxed{0}\boxed{1}$ | 2 = $\boxed{1}\boxed{0}$ | 3 = $\boxed{1}\boxed{1}$ |
| Background Color | Color in Reg. 0 | Color in Reg. 1 | Color in Reg. 2 |

This is why these modes allow four possible colors for each pixel. If the value of the two bits is 0, the pixel is the color of the background. If the value is 1, the pixel is the color in register 0. A value of 2 gives the color in register 1, and a value of 3 gives the color in register 2. (This resembles the use of the COLOR statement in BASIC). Color register 3 does not get used in these modes.

Since there are 8 bits in a byte, and only two bits are needed to specify the color of a pixel in Graphics modes 3, 5, and 7, we get 4 pixels on the screen for each byte in screen RAM. Internal Representation Worksheet #4 will provide you with a better understanding of this coding scheme.

# Internal Representation Worksheet #4

In Graphics mode 8, each bit in a byte corresponds to one pixel. A byte has 8 bits and so each byte codes the information in screen RAM for 8 pixels. In graphics modes 3, 5, and 7 two bits are needed for each pixel and so each byte provides the information for only four pixels. Type the following:

```
GRAPHICS 3
SCRAM=256*PEEK(89)+PEEK(88)
SCRAM=SCRAM+10:POKE SCRAM,0
```

Just as you did in Worksheet #3, use the cursor control arrows to change the zero in "POKE SCRAM,0" to the decimal numbers in the following list. Notice the correspondence between the value of every two bits in the binary representation of the number being POKEd into screen memory and the colors that light up on the screen.

| Decimal (Number to POKE) | Binary representation | Value of every two bits (COLOR) |
|---|---|---|
| 255 | 11 11 11 11 | 3 3 3 3 |
| 204 | 11 00 11 00 | 3 0 3 0 |
| 51 | 00 11 00 11 | 0 3 0 3 |
| 108 | 01 10 11 00 | 1 2 3 0 |
| 109 | 01 10 11 01 | 1 2 3 1 |

Now change the 10 to a 1 in the BASIC statement so that it reads:

```
SCRAM=SCRAM+1:POKE SCRAM,0
```

Try POKEing some different numbers into screen memory and see the effect. Also try writing FOR loops to POKE different numbers into different places in screen memory (see Worksheet #2). Finally, experiment with this technique in both graphics modes 5 and 7 to see the differences in resolution.

The amount of screen RAM needed for a full graphics screen varies among modes 3, 5, and 7 because the resolution is different among these modes. The chart below calculates the number of bytes needed for each of these modes. Remember that only one byte is required for every four pixels.

| | Mode 3 | Mode 5 | Mode 7 |
|---|---|---|---|
| Horizontal resolution (# of pixels) | 40 | 80 | 160 |
| Bytes needed for each row | 10 | 20 | 40 |
| Vertical resolution (# of rows) | 24 | 48 | 96 |
| Total # of bytes (screen RAM) | 240 | 960 | 3840 |

NOTE: Since there are 10 bytes per row in Graphics 3, SCRAM+SCRAM+10 was used in Worksheet #4 to bring the graphics down exactly one row.

Mode 7 uses up quite a lot of memory -- almost 4K. So it was decided that there might be situations when programmers would want the same resolution as Graphics mode 7 (40 pixels per row; 96 rows), but they wouldn't want to use so much memory. The only way to do this is to make only one bit in each byte correspond to a pixel on the screen. Then every byte can store the information for 8 pixels rather than only 4 pixels as in Graphics 7. This is exactly what happens in Graphics mode 6.

Thus, in Graphics 6 you get twice as many pixels for every byte. Therefore, you use only half as much memory as in Graphics 7. There is of course a price to pay. When only one bit is used for every pixel, there can be only two possibilities for each pixel -- either it is on (the bit is 1) or it is off (the bit is 0). This means that you only get one color -- the color in register zero.

Similarly, Graphics 4 has the same resolution as Graphics 5, but uses only half the screen RAM. Again you get only two colors rather than four -- either color register 0 or the background.

You might wish to try poking some numbers into screen RAM in these two modes to be sure you understand how these modes work. For example, use the following routine to POKE 109 into screen RAM and observe the dfference between the Graphics 4 and the Graphics 5 display. Type:

```
GRAPHICS 4
SCRAM = 256*PEEK(89)+PEEK(88)
SCRAM = SCRAM+10:POKE SCRAM,109
```

Repeat this set of instructions for Graphics 5.

## Graphics Modes 0, 1, and 2

These graphics modes are used to print text on the screen and are a little more mysterious in how the computer internally represents the characters you see on the screen. In this section you'll learn about the coding techniques used in the text graphics modes.

The most difficult concept to understand, is how the computer puts a character up on the screen. One might say that the computer actually "plots" a character on the screen. It does this in the same way as pixels appear in Graphics mode 8, where each bit that is a 1 turns on a corresponding pixel on the screen. Consider Diagram 5. These are the eight bytes used to represent the letter "A". The "A" is formed by the ones in the bytes.

Diagram 5

| Byte | Binary | Decimal |
|---|---|---|
| 1 | 00000000 | 0 |
| 2 | 00011000 | 24 |
| 3 | 00111100 | 60 |
| 4 | 01100110 | 102 |
| 5 | 01100110 | 102 |
| 6 | 01111110 | 126 |
| 7 | 01100110 | 102 |
| 8 | 00000000 | 0 |

Internal Representation Worksheet #5 develops this idea.

## Internal Representation Worksheet #5

The decimal values of the eight bytes used to represent an "A" shown in Diagram 1 are: 0, 24, 60, 102, 102, 126, 102, and 0. Type the following routine to see that these numbers really make an "A".

```
GRAPHICS 8
SETCOLOR 2,0,0
SCRAM=256*PEEK(89)+PEEK(88)+5
POKE SCRAM,0:SCRAM=SCRAM+40
```

Five is added to the screen RAM location, in order to move the display five spaces to the right so that it is easier to see. Use the CTRL and the arrow keys to move the cursor up to the POKE statement and change the zero to 24 (you'll need to press the CTRL and the INSERT keys to get an extra space), then press RETURN. Do this again for the rest of the numbers: 60, 102, 102, 126, 102, and 0. You should see the "A" in the upper left hand corner of the screen.

The computer keeps all of the values of the eight bytes necessary for "drawing" every character stored away in its memory. In fact, you can actually PEEK into memory to find those numbers. Try the following:

```
Press SYSTEM RESET
CHBAS=756
```

This sets CHBAS equal to the address which holds the high order byte of where the character data is stored.

```
CHARS=256*PEEK(CHBAS)
```

Every eight bytes starting at this address provide the bit pattern for a different character.

```
A=CHARS+8*33
```

An "A" is the 33rd character in the character set. Actually there are 33 characters before the "A" because there's a character numbered zero. Since each character's shape takes 8 bytes to define, the bit pattern for the letter "A" starts 8*33 bytes after the beginning of the character set.

```
FOR I=0 TO 7:PRINT PEEK(A+I):NEXT I
```

This prints the 8 bytes used to define an "A".

Try this:

```
B=CHARS+8*34
FOR I=0 TO 7:PRINT PEEK(B+I):NEXT I
```

Let's use these numbers more directly to put letters on a Graphics mode 8 screen as we did earlier. Try the following:

```
GRAPHICS 8
SETCOLOR 2,0,0
SCRAM=256*PEEK(89)+PEEK(88)
CHARS=256*PEEK(756)
A=CHARS+8*33:B=CHARS+8*34
FOR I=0 TO 7:POKE SCRAM+40*I,PEEK(A+I):NEXT I
FOR I=0 TO 7:POKE SCRAM+1+40*I,PEEK(B+I):NEXT I
```

And for something a little more interesting, run this program:

Press SYSTEM RESET

```
100 GR. 8
110 SETCOLOR 2,0,0
120 SCRAM=256*PEEK(89)+PEEK(88)
130 CHARS=256*PEEK(756)
140 COLUMN=0
150 FOR CH=0 TO 127
160 FOR I=0 TO 7
170 POKE SCRAM+CH+40*I,PEEK(CHARS+8*CH+I)
180 NEXT I
190 COLUMN=COLUMN+1
200 IF COLUMN=40 THEN SCRAM=SCRAM+400:COLUMN=0
210 NEXT CH
220 END
```

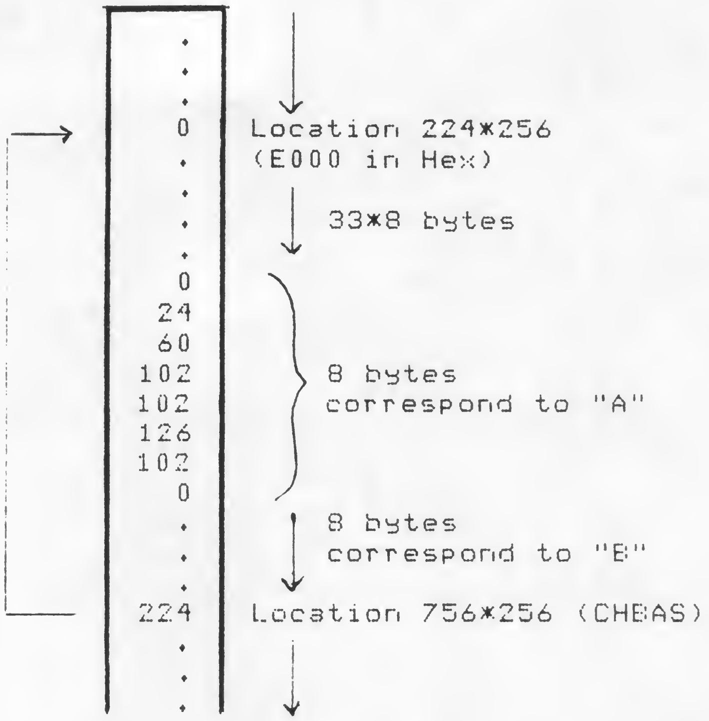
There you have the entire character set copied to your Graphics 8 screen. Play with this technique for a while if you like. Try things like making your letters upside down by copying the bytes in reverse order as in the example lines listed below.

```
SCRAM=SCRAM+400
FOR I=0 TO 7:POKE SCRAM+40*I,PEEK(CHARS+8*33+7-I):NEXT I
```

This assumes that SCRAM and CHARS are defined from running the program above.

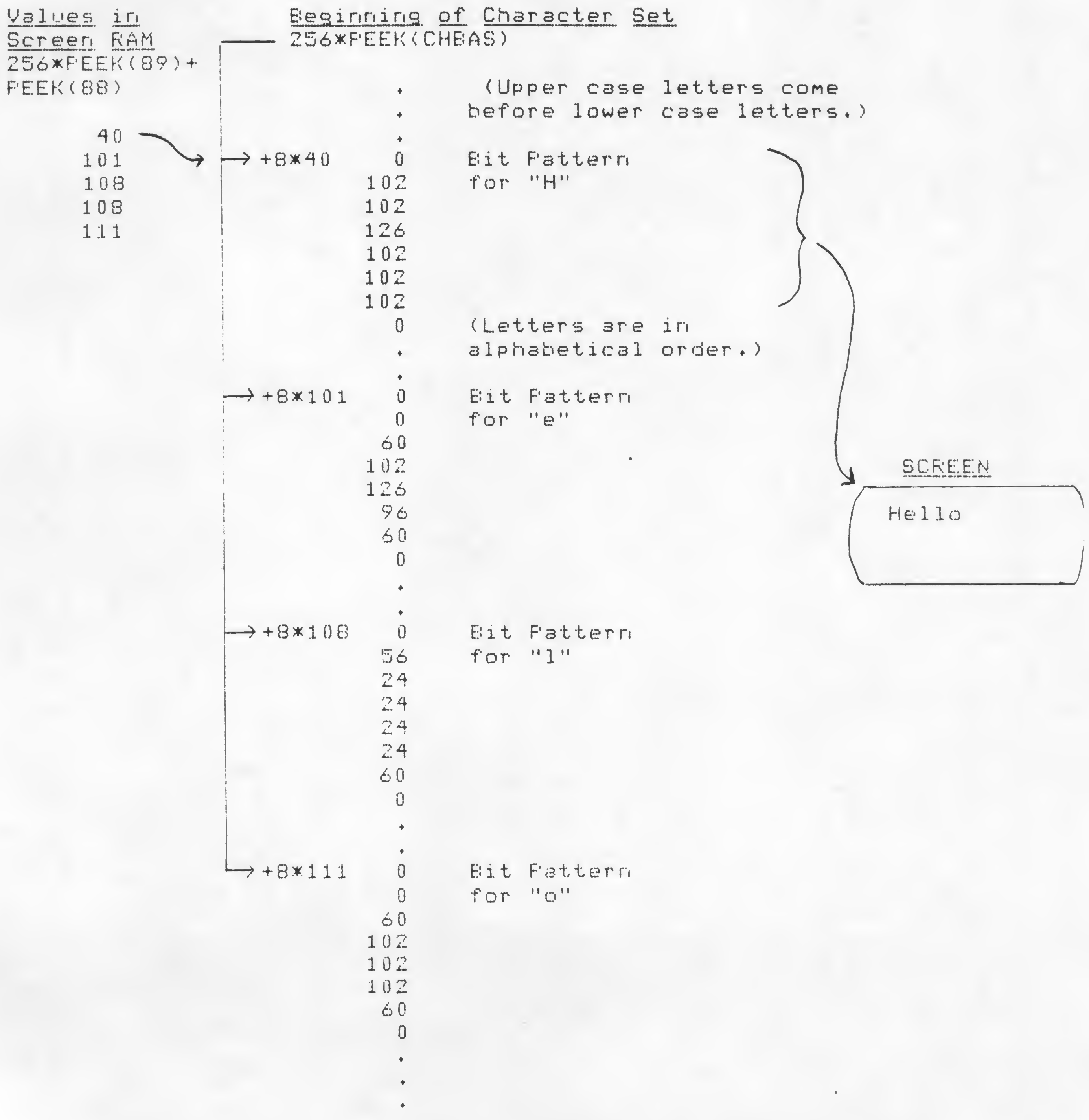
You've learned that each character is stored as an eight byte bit pattern representing the character's shape, and that the eight bytes corresponding to each character are stored in a long list in memory (see Diagram 6). The base address for the character set is 224*256 in decimal or $E000 in hexadecimal (base 16).

<u>Diagram 6</u>



By now you may realize that the number 33 and the letter "A" have a special relationship as do the number 34 and "B", etc. These numbers (33 and 34) are called <u>offsets</u>. They provide the computer with a way to find the eight bytes it needs to put any particular character on the screen. For example, character #33 (the letter "A") is always found 33*8 bytes (since each character takes up 8 bytes) past the beginning of the character set list. These offset numbers (often referred to as the <u>internal character set</u>) are the very numbers that are placed in screen RAM as the code for each character that is to be put on the screen. See Diagram 7 on the next page and the Internal Character Set Chart at the back of this module.

Diagram 7


Values in Screen RAM
256*PEEK(89)+
PEEK(88)
40
101
108
108
111
Beginning of Character Set
256*PEEK(CHBAS)
(Upper case letters come before lower case letters.)
+8*40
0
102
102
126
102
102
102
0
Bit Pattern for "H"
(Letters are in alphabetical order.)
+8*101
0
0
60
102
126
96
60
0
Bit Pattern for "e"
+8*108
0
56
24
24
24
24
60
0
Bit Pattern for "l"
+8*111
0
0
60
102
102
102
60
0
Bit Pattern for "o"
SCREEN
Hello

In the text graphics modes, each byte in screen RAM represents one character on the screen. The value of the byte provides the computer with the character number (this, by the way, is not the same number as the character's ATASCI value) which is used to look up a bit pattern (8 bytes) in the character set table.

This technique is used in Graphics modes 0, 1, and 2, but Graphics modes 1 and 2 have a slight variation so as to provide colored characters. Worksheet #6 explores this further.

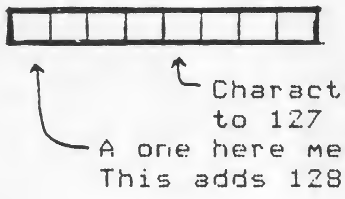
## Internal Representation Worksheet #6

There are 128 different characters numbered from 0 to 127. But each byte in screen RAM is used to represent one character, and a byte can be a number from 0 to 255. With only 128 different characters, what does one get when the value of a byte in screen memory is greater than 127? Try the following to discover the answer.

```
Press SYSTEM RESET
SCRAM=256*PEEK(89)+PEEK(88)
FOR I=0 TO 255:POKE SCRAM+I,I:NEXT I
```

There you have it! Bytes with values from 128 to 255 give the same characters as those numbered 0 to 127, but they are in inverse video. Actually seven bits in each byte are used to provide the character number in Graphics 0 and one bit tells the computer if the character should be displayed in inverse video.

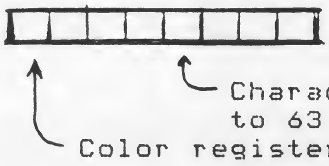
GR. 0 Byte


Character # from 0 (0000000) to 127 (1111111).
A one here means inverse video. This adds 128 to the value of the byte.

```
Press SYSTEM RESET
POKE SCRAM,33
POKE SCRAM+1,33+128
```

In Graphics modes 1 and 2 only six bits are used to provide the character number so that the two extra bits can be used to give a color register number.

GR. 1 or 2 byte


Character # from 0 (000000) to 63 (111111).
Color register # from 0 (00) to 3 (11).

Since only six bits are used for the character number in these modes, you can only get the first 64 characters in the character set. Let's explore this.

```
GRAPHICS 1
SCRAM=256*PEEK(89)+PEEK(88)
FOR I=0 TO 63:POKE SCRAM,I:SCRAM=SCRAM+1:NEXT I
```

This gives the first 64 characters with the color stored in color register 0 because the first two bits of all the bytes are zero.

```
FOR I=0 TO 63:POKE SCRAM,I+64:SCRAM=SCRAM+1:NEXT I
```

Adding 64 to each byte sets the first two bits to 01.

```
FOR I=0 TO 63:POKE SCRAM,I+128:SCRAM=SCRAM+1:NEXT I
```

Now you get color register 2 because the first two bits are 10 which is 2 in decimal.

```
FOR I=0 TO 63:POKE SCRAM,I+128+64:SCRAM=SCRAM+1:NEXT I
```

And finally color register 3.

In Graphics modes 1 and 2, if you PRINT #6;"AaAa" (type the last two letters in inverse video) you get all capital A's on the screen but in four different colors. Can you figure out why this works?

## Summary and Challenges

Recall that the value stored in memory location 756 (CHBAS) tells the computer where to find the list of bytes which provide the pattern of bits for each character. You can change this value so that the computer looks elsewhere for those 8 byte patterns. Worksheet #7 will help you learn to use this fact to define your own characters.

Below is a summary of the use of each of the bits in a byte placed in screen RAM for each graphics mode. The bits are referred to as follows:

Bit Positions: | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

| Graphics mode | Use of Byte in Screen RAM |
|---|---|
| 0 | Bits 0 - 6 code a value from 0 to 127 which is used as an offset into the character set table (8 bytes per char). If Bit 7 is a one, you get inverse video. |
| 1, 2 | Bits 0 - 5 code a value from 0 to 63 which is used as an offset into the character set table. Bits 6 and 7 provide the color register. |
| 3, 5, 7 | Every two bits provide the color register (0 - 2 or background) for each pixel. |
| 4, 6 | Each on bit is a pixel whose color is in register 0. |
| 8 | Each on bit is a pixel whose hue is in register 2 and whose luminance comes from color register 1. |

## Internal Representation Worksheet #7

Type in and run the following program:

```
10 CHARS=256*PEEK(756)
20 CHSET=120*256
```

We'll put our altered character set beginning at CHSET. First you'll need to copy the computer's character set to this area of memory so that we can change it.

```
30 FOR I=0 TO 127*8+7
40 POKE CHSET+I,PEEK(CHARS+I)
50 NEXT I
```

The next statement tells the computer where to find our character set.

```
60 POKE 756,120
70 END
```

It will take a minute or two for the data to be transferred. After running the above program, you can change any or all of the characters in the character set by POKEing into memory a new 8 byte pattern for the character. For example, a new "A" can be designed:

```
00000000      0
00111100     60
00111100     60   These are decimal values
01011010     90   of each of the bytes on the left.
01011010     90
01011010     90
10011001    153
00000000      0
```

The letter "A" is the 33rd character in the character set. Therefore, its bit pattern starts at CHSET+8*33.

```
POS=CHSET+8*33
POKE POS,0
POKE POS+1,60
POKE POS+2,60
POKE POS+3,90
POKE POS+4,90
POKE POS+5,90
POKE POS+6,153
POKE POS+7,0
```

Notice that every letter "A" on your screen changes as you POKE in a new bit pattern.

Play with this idea and see what interesting things you can do. Try redesigning the blank (character number 0). For example, try the following:

```
POKE CHSET,255
POKE CHSET+1,255
POKE CHSET+2,255
POKE CHSET+5,255
POKE CHSET+6,255
POKE CHSET+7,255
```

Create your own shape and define it as one of the letters in the character set.

You might try using the APX program called INSTEDIT which enables you to design new characters much more easily. There are copies of INSTEDIT in the camp software library.